The Quantum IO Monad

QIO

Alexander S. Green
asg@cs.nott.ac.uk

Foundations of Programming Group,
School of Computer Science & IT,
University of Nottingham
Introduction

- We would like to model Quantum Computations.
We would like to model Quantum Computations.

The QIO Monad, can be thought of as a register of Qubits that plugs into your classical computer.
Introduction

- We would like to model Quantum Computations.
- The QIO Monad, can be thought of as a register of Qubits that plugs into your classical computer.
- It provides a framework for constructing quantum computations...
We would like to model Quantum Computations.
The QIO Monad, can be thought of as a register of Qubits that plugs into your classical computer.
It provides a framework for constructing quantum computations...
... and simulates the running of these computations.
Haskell and Monads

- Haskell is a pure functional programming language, so any computations that may involve side effects make use of Monads.
• Haskell is a **pure** functional programming language, so any computations that may involve side effects make use of Monads.

• Monads are defined by a *return* function, and a bind function denoted (\( \_\_\_\_\_ \))
Haskell is a pure functional programming language, so any computations that may involve side effects make use of Monads.

- Monads are defined by a `return` function, and a bind function denoted `(>>=`)

```haskell
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
```
Haskell and Monads

• Haskell is a pure functional programming language, so any computations that may involve side effects make use of Monads.

• Monads are defined by a \texttt{return} function, and a bind function denoted ( \texttt{>>=} )

\[
\text{class } \texttt{Monad } m \ \texttt{where} \\
(\gg=) :: m a \to (a \to m b) \to m b \\
\texttt{return} :: a \to m a
\]

• Haskell provides the \texttt{do} notation to make monadic programming easier.
'do' notation

- IO in Haskell takes place in the IO Monad.
‘do’ notation

• IO in Haskell takes place in the IO Monad.
• For example, echoing a character to the screen

\[
\text{getChar :: IO Char}
\]
\[
\text{putChar :: Char \rightarrow IO ()}
\]
'do' notation

- IO in Haskell takes place in the IO Monad.
- For example, echoing a character to the screen
  
  ```haskell
  getChar :: IO Char
  putChar :: Char → IO ()
  
  echo :: IO ()
  echo = getChar >>= (λc → putChar c) >>= echo
  ```
’do’ notation

- IO in Haskell takes place in the IO Monad.
- For example, echoing a character to the screen

  \[
  \text{getChar} :: \text{IO} \ \text{Char} \\
  \text{putChar} :: \text{Char} \to \text{IO} ()
  \]

- \( \text{echo} :: \text{IO} () \)

  \[
  \text{echo} = \text{getChar} \gg (\lambda c \rightarrow \text{putChar} \ c) > > \text{echo}
  \]

- or in \text{do} notation
’do’ notation

- IO in Haskell takes place in the IO Monad.
- For example, echoing a character to the screen
  \[\text{getChar} :: \text{IO Char}\]
  \[\text{putChar} :: \text{Char} \to \text{IO ()}\]

  \[\text{echo} :: \text{IO ()}\]
  \[\text{echo} = \text{getChar} \gg\gg (\lambda c \to \text{putChar } c) > > \text{echo}\]

- or in do notation
  \[\text{echo} = \text{do } c \leftarrow \text{getChar}\]
  \[\text{putChar } c\]
  \[\text{echo}\]
The QIO Monad

- The QIO Monad has been designed so that Quantum computations can be defined within Haskell.
The QIO Monad

- The QIO Monad has been designed so that Quantum computations can be defined within Haskell.
- The `do` notation provided by Haskell is very useful for this purpose.
The QIO Monad

- The QIO Monad has been designed so that Quantum computations can be defined within Haskell.
- The do notation provided by Haskell is very useful for this purpose.

\[
\text{trueBit} :: \text{QIO Boolean} \\
\text{trueBit} = \text{do} \ q_b \leftarrow \text{mkQbit True} \\
\quad x \leftarrow \text{measQbit} \ q_b \\
\quad \text{return} \ x
\]
API

- What can we do in the QIO Monad?
What can we do in the QIO Monad?

\[ mkQbit :: \text{Bool} \rightarrow QIO \ Qbit \]
What can we do in the QIO Monad?

- \( mkQbit :: \text{Bool} \rightarrow QIO \ Qbit \)

When initialising a qubit the user must define which of the base states, True or False (\(|0\rangle\) or \(|1\rangle\)), to initialise it into.
What can we do in the QIO Monad?

- \( \text{mkQbit} :: \text{Bool} \rightarrow \text{QIO Qbit} \)

- When initialising a qubit the user must define which of the base states, True or False (\(|0\rangle\) or \(|1\rangle\)), to initialise it into.

- \( \text{measQbit} :: \text{Qbit} \rightarrow \text{QIO Bool} \)
What can we do in the QIO Monad?

- `mkQbit :: Bool → QIO Qbit`

  When initialising a qubit the user must define which of the base states, True or False (`|0⟩` or `|1⟩`), to initialise it into.

- `measQbit :: Qbit → QIO Bool`

  The measurement of a qubit always results in a boolean value.
• What can we do in the QIO Monad?

  \[ \text{mkQbit} :: \text{Bool} \rightarrow \text{QIO Qbit} \]

• When initialising a qubit the user must define which of the base states, True or False (\(|0\rangle\) or \(|1\rangle\)), to initialise it into.

  \[ \text{measQbit} :: \text{Qbit} \rightarrow \text{QIO Bool} \]

• The measurement of a qubit always results in a boolean value.

• What else can be done with these qubits?
Unitaries.

- It is possible to construct unitary operators, and apply them to the relevant qubits.
Unitaries.

- It is possible to construct unitary operators, and apply them to the relevant qubits.

  \[ \text{applyU} :: U \rightarrow QIO () \]
Unitaries.

- It is possible to construct unitary operators, and apply them to the relevant qubits.

  \[ \text{apply} U :: U \rightarrow QIO () \]

- There are 5 unitary constructors that are available:
Unitaries.

- It is possible to construct unitary operators, and apply them to the relevant qubits.
- \( \text{applyU} :: U \rightarrow QIO () \)
- There are 5 unitary constructors that are available:
- \( \text{unot} :: Qbit \rightarrow U \)
  which will rotate the given qubit by \(180^\circ\) as in the Not rotation.
Unitaries.

- It is possible to construct unitary operators, and apply them to the relevant qubits.

  \( \text{applyU} :: U \rightarrow QIO () \)

- There are 5 unitary constructors that are available:

  \( \text{unot} :: Qbit \rightarrow U \)

  which will rotate the given qubit by 180° as in the Not rotation.

  \[
  \begin{pmatrix}
  0 & 1 \\
  1 & 0
  \end{pmatrix}
  \]
Unitaries..

- \( \text{uhad} :: Qbit \rightarrow U \)
  which will rotate the given qubit by \( 90^\circ \) as in the Hadamard rotation.
Unitaries..

- $\text{uhad} :: \text{Qbit} \rightarrow U$
  which will rotate the given qubit by $90^\circ$ as in the Hadamard rotation.

- $\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
Unitaries..

- $uhad :: Qbit \rightarrow U$
  which will rotate the given qubit by $90^\circ$ as in the Hadamard rotation.

- $\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

- $uphase :: Qbit \rightarrow Float \rightarrow U$
  which will rotate the given qubit by the given phase change ($\phi$).
Unitaries..

- \( \text{uhad} :: Qbit \rightarrow U \)
  which will rotate the given qubit by \( 90^\circ \) as in the Hadamard rotation.

- \( \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \)

- \( \text{uphase} :: Qbit \rightarrow Float \rightarrow U \)
  which will rotate the given qubit by the given phase change (\( \phi \)).

- \( \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i \phi} \end{pmatrix} \)
Unitaries...

- $\text{swap} :: \text{Qbit} \rightarrow \text{Qbit} \rightarrow U$

which simply swaps the two given qubits.
Unitaries...

- \( \text{swap} :: \text{Qbit} \rightarrow \text{Qbit} \rightarrow U \)
  which simply swaps the two given qubits.

- \( \text{cond} :: \text{Qbit} \rightarrow (\text{Bool} \rightarrow U) \rightarrow U \)
  which given a control qubit, will conditionally do the corresponding unitary given by the function.
  (The control qubit must not be effected by the unitaries)
Unitaries...

- \( \text{swap} :: Qbit \rightarrow Qbit \rightarrow U \)
  which simply swaps the two given qubits.

- \( \text{cond} :: Qbit \rightarrow (Bool \rightarrow U) \rightarrow U \)
  which given a control qubit, will conditionally do the corresponding unitary given by the function.
  (The control qubit must not be effected by the unitaries)

- It is this conditional operation that can be used to entangle qubits.
Unitaries...

- \(\text{swap} :: \text{Qbit} \rightarrow \text{Qbit} \rightarrow U\)
  which simply swaps the two given qubits.

- \(\text{cond} :: \text{Qbit} \rightarrow (\text{Bool} \rightarrow U) \rightarrow U\)
  which given a control qubit, will conditionally do the corresponding unitary given by the function. (The control qubit must not be effected by the unitaries)

- It is this conditional operation that can be used to entangle qubits.

- The \(U\) datatype of unitaries, also forms a **Monoid** meaning there is an append operation for combining unitaries sequentially.
Running Quantum Computations?

- Along with creating quantum computations, the QIO Monad also provides two ways of evaluating them.
Along with creating quantum computations, the QIO Monad also provides two ways of evaluating them.

\[ \text{run} :: QIO \ a \rightarrow IO \ a \]
Along with creating quantum computations, the QIO Monad also provides two ways of evaluating them.

\[
\text{run} :: QIO \ a \rightarrow IO \ a
\]

Running a quantum computation returns a probabilistic result for each measurement.
Along with creating quantum computations, the QIO Monad also provides two ways of evaluating them.

- **run**: \( \text{run} :: QIO \ a \rightarrow IO \ a \)
- **Running a quantum computation returns a probabilistic result for each measurement.**
- **sim**: \( \text{sim} :: QIO \ a \rightarrow \text{Prob} \ a \)
Along with creating quantum computations, the QIO Monad also provides two ways of evaluating them.

- \( \text{run} :: QIO \ a \to IO \ a \)

- Running a quantum computation returns a probabilistic result for each measurement.

- \( \text{sim} :: QIO \ a \to Prob \ a \)

- Simulating a quantum computation returns a probability distribution of all the possible measurement outcomes.
Along with creating quantum computations, the QIO Monad also provides two ways of evaluating them.

- \( \text{run} :: \textit{QIO} \ a \rightarrow \textit{IO} \ a \)

- Running a quantum computation returns a probabilistic result for each measurement.

- \( \text{sim} :: \textit{QIO} \ a \rightarrow \textit{Prob} \ a \)

- Simulating a quantum computation returns a probability distribution of all the possible measurement outcomes.

- We would also like to be able to display the internal state of the system at any time, possibly by showing the complex amplitudes for each base state.
Computations.

\[
\begin{align*}
qPlus &:: QIO Qbit \\
qPlus &= \textbf{do} \ qa \leftarrow \text{mkQbit} \ False \\
&\quad \text{applyU (uhad qa)} \\
&\quad \text{return qa} \\
\text{randBit} &:: QIO \ Bool \\
\text{randBit} &= \textbf{do} \ qa \leftarrow qPlus \\
&\quad x \leftarrow \text{measQbit} \ qa \\
&\quad \text{return} \ x
\end{align*}
\]
share :: Qbit → QIO Qbit
share qa = do qb ← mkQbit False
    applyU (cond qa (λa → if a then (unot qb) else mempty))
    return qb

bell :: QIO (Qbit, Qbit)
bell = do qa ← qPlus
    qb ← share qa
    return (qa, qb)
test\_bell :: QIO (Bool, Bool)
test\_bell = do qb ← bell
               b ← measQ qb
               return b
Teleportation.

alice :: Qbit → Qbit → QIO (Bool, Bool)
alice aq bsq = do applyU (cond aq

(λa → if a then (unot bsq)
else mempty))
applyU (uhad aq)

cd ← measQ (aq, bsq)
return cd
Teleportation..

\[\text{uZ} :: \text{Qbit} \rightarrow \text{U} \]
\[\text{uZ} \ \text{qb} = (\text{uphase} \ \text{qb} \ 0.5)\]

\[\text{bobsU} :: (\text{Bool}, \text{Bool}) \rightarrow \text{Qbit} \rightarrow \text{U} \]
\[\text{bobsU} (\text{False}, \text{False}) \ \text{qb} = \text{mempty} \]
\[\text{bobsU} (\text{False}, \text{True}) \ \text{qb} = (\text{unot} \ \text{qb}) \]
\[\text{bobsU} (\text{True}, \text{False}) \ \text{qb} = (\text{uZ} \ \text{qb}) \]
\[\text{bobsU} (\text{True}, \text{True}) \ \text{qb} = ((\text{unot} \ \text{qb}) \ \text{‘mappend‘} (\text{uZ} \ \text{qb})) \]

\[\text{bob} :: \text{Qbit} \rightarrow (\text{Bool}, \text{Bool}) \rightarrow \text{QIO} \ \text{Qbit} \]
\[\text{bob bsq cd} = \text{do} \ \text{applyU} (\text{bobsU} \ cd \ bsq) \]
\[\text{return} \ \text{bsq} \]
teleportation :: Qbit → QIO Qbit

\[
\text{teleportation } iq = \text{do } (bsq_1, bsq_2) \leftarrow \text{bell} \\
    cd \leftarrow \text{alice } iq \ bsq_1 \\
    tq \leftarrow \text{bob } bsq_2 \ cd \\
\text{return } tq
\]
• There is a symmetry between initialising a qubit, and measuring a qubit.
• There is a symmetry between initialising a qubit, and measuring a qubit.
• Larger quantum data structures can be defined using qubits, in the same way classical data structures are defined using bits.
• There is a symmetry between initialising a qubit, and measuring a qubit.

• Larger quantum data structures can be defined using qubits, in the same way classical data structures are defined using bits.

• We have defined a class of quantum data types, \( Qdata \) For which an \( mkQ \) initialisation function and a \( measQ \) measurement function must be defined, between the quantum datatype and its classical counter-part.
There is a symmetry between initialising a qubit, and measuring a qubit.

Larger quantum data structures can be defined using qubits, in the same way classical data structures are defined using bits.

We have defined a class of quantum data types, \textit{Qdata}.
For which an \textit{mkQ} initialisation function and a \textit{measQ} measurement function must be defined, between the quantum datatype and its classical counter-part.

\textbf{instance} \ \textit{Qdata Bool Qbit where}

\begin{itemize}
  \item \textit{mkQ} = \textit{mkQbit}
  \item \textit{measQ} = \textit{measQbit}
\end{itemize}
instance (Qdata a qa, Qdata b qb) ⇒ Qdata (a, b) (qa, qb) where

mkQ (a, b) = do qa ← mkQ a
                qb ← mkQ b
                return (qa, qb)

measQ (qa, qb) = do a ← measQ qa
                    b ← measQ qb
                    return (a, b)
Further Work

- We are going to implement some of the famous quantum algorithms, such as Shor’s quantum factorisation algorithm.
Further Work

- We are going to implement some of the famous quantum algorithms, such as Shor’s quantum factorisation algorithm.
- We are going to use the QIO Monad to start reasoning about quantum computation in general.
Further Work

- We are going to implement some of the famous quantum algorithms, such as Shor’s quantum factorisation algorithm.
- We are going to use the QIO Monad to start reasoning about quantum computation in general.
- We are going to model other forms of quantum computer within the QIO Monad, such as the Measurement based model of quantum computations.
Further Work

- We are going to implement some of the famous quantum algorithms, such as Shor’s quantum factorisation algorithm.
- We are going to use the QIO Monad to start reasoning about quantum computation in general.
- We are going to model other forms of quantum computer within the QIO Monad, such as the Measurement based model of quantum computations.
- Thank you all for listening!